A user-centric approach for developing and deploying service front-ends in the future internet of services

David Lizcano* and Javier Soriano

School of Computing
Universidad Politécnica de Madrid
Campus de Montegancedo

Marcos Reyes and Juan J. Hierro

Telefónica Investigación y Desarrollo

Abstract: Service-Oriented Architectures (SOAs) based on web services have attracted a great deal of interest and Internet Technology (IT) investment over the last few years, principally in the context of business-to-business integration within corporate intranets. However, they are now evolving and breaking through enterprise boundaries in a revolutionary attempt to make the approach pervasive. This is leading to what we call a user-centric SOA. A user-centric SOA is an SOA conceived as an internet of services made up of compositional resources empowering end users to collaboratively remix and ubiquitously exploit these resources. In this paper we explore the architectural basis, technologies, frameworks and tools considered necessary to tackle this novel vision of SOA. We also present the rationale behind EzWeb/FAST, an ongoing EU-funded project whose first outcomes could serve as a preliminary proof of concept.

Keywords: service-oriented architecture; SOA; web services; mashup; web 2.0; user-centric SOA; internet of services; composite applications; user-centric SOAs and their front-ends in the future internet of services.

Biographical notes: David Lizcano holds an MSc degree with honours in Computer Science (2006) from the Universidad Politécnica de Madrid (UPM), Spain. He is a PhD student at the UPM's Department of Computer Science, and holds a research grant from the UPM and the European Social Fund under

their Research Personnel Training programme. He is working as a Research Associate and PhD candidate at the Computer Networks and Web Technologies Lab (CoN-WeT Lab, UPM), where he is currently involved in several national and European-funded projects relating to web 2.0 and Enterprise 2.0 technologies and Service-Oriented Architectures (SOA).

Dr. Javier Soriano holds an MSc in Computing (1998) and a PhD in Computer Science (2003) from the Universidad Politécnica de Madrid (UPM), Spain. He is an Associate Professor at the UPM's School of Computer Science and leads the Computer Networks and Web Technologies Lab (CoNWeT Lab). He is a member of the UPM's Information and Communications Technology Research Group (ICTG-CETTICO) and collaborates with the Centre of Computing and Communications Technology Transfer (CETTICO). He is a co-author of over 30 journal publications and conference articles and of several books. He is also involved with the Networked European Software & Services Initiative (NESSI).

Marcos Reyes Urefia holds a Master's degree with honours in Computer Science from the Universidad Politécnica de Madrid, Spain. Since 2000, Urefia has been with Telefonica I+D, where he has been involved in several R&D projects related to identity management and BPM systems applied to workforce management and trouble-ticketing systems. In 2005 Urefia was promoted to chief architect and designer of Telefonica I+D's SmartFlow Platform, FAST, EzWeb and several Enterprise 2.0 projects by means of applying Web 2.0 concepts to the enterprise systems, all of them within the Morfeo Open Source community (http://www.morfeo-project.org) created by Telefonica. Besides these, Urefia coordinates several organizative matters of the community.

Juan J. Hierro currently holds the position of Chief Technology Expert on Software Technologies at Telefonica I+D, Spain. He obtained a degree in Computer Science in 1990 from the Universidad Politécnica de Madrid. He is involved in R&D projects related to middleware technologies and operation support systems. He has actively represented the company in OMG and OpenGroup and has been deeply involved in the definition of relevant standards, including CORBA. Since January 2004 he has been responsible for innovation in the field of basic software technologies for services and network management. He chairs the open source community Morfeo created by Telefonica.

1 Introduction

Service-Oriented Architectures (SOAs) have attracted a great deal of interest over the last few years. SOAs increase asset reuse, reduce integration expenses and improve businesses' agility in responding to new demands (Alonso *et al.*, 2004). Nonetheless, until now, mainstream SOA research and development has focused mainly on middleware and scalability, service engineering and automating service composition using Business Process Management (BPM) technologies (MacKenzie, 2006). Little or no attention has been paid to service front-ends, which we regard as a fundamental part of a SOA (Schroth and Christ, 2007). Consequently, SOAs are confined to a technical layer hidden to the end user. The evolution of web-based interfaces is a testimony to the progress achieved in improving service usability. However, the existing web-based service front-ends are far from completely meeting end-user expectations (Davenport,

2005). Applications and information portals are still based on monolithic, inflexible, non-context-aware, noncustomisable and unfriendly User Interfaces (UIs) (OASIS, 2007). As a consequence, end users do not really benefit from the advantages promoted by service orientation in terms of modularity, flexibility and composition (McAfee, 2005). In addition, service front-ends are constructed in an *ad hoc* manner and without formal engineering methods and tools that could speed up the time to market. This vision is the first result of the Service Front End (SFE) Open Alliance initiative. The SFE Open Alliance aims to integrate results from several relevant R&D projects in the field to produce open specifications and an open source reference implementation of the components of an envisioned web platform to access service content and objects in the future internet (Lizcano *et al.*, 2008). Our approach is supported by two hot research projects: FAST and EzWeb. These projects are referenced and examined at length in this paper. The rationale behind FAST, a complex-gadget development environment, and EzWeb, a reference architecture and implementation of an open Enterprise 2.0 mashup platform, is both presented and exploited in a use case as proof of concept. These two elements together empower users to co-produce and share instant composite applications and their components (OASIS, 2003). The remainder of the paper is structured as follows: First we revisit the notion of traditional WS-SOA-based composite applications, and analyse their major shortcomings with regard to the ideal user-service interaction in a user-centred internet of services (Section 2). Also, we elaborate on a use case that illustrates the current user-service interaction needs in composite application development, where current IT like traditional web services or novel mashup ideas based on disparate and independent gadgets have more than once been found wanting. The above shortcomings of the current approaches and technologies and the ideal solutions for these problems can be easily identified in this use case (Section 3). We go on in Section 4 to illustrate the guiding principles towards our goal. We then present a novel architecture framework built on the FAST development approach and the EzWeb exploitation platform. FAST creates the building blocks and EzWeb interconnects these building blocks to compose instant applications (Section 5). Sections 6 and 7 explain the technical details of FAST and EzWeb, respectively. They give a better understanding of the two platforms. Section 8 describes the application of FAST and EzWeb as a proof of concept, dealing with the use case presented in Section 3. Section 9 includes an evaluation of the framework created using both FAST and EzWeb. It compares the framework with other state-of-the-art solutions, thereby clearly identifying our contribution. Section 10 presents other related work and, finally, the last section concludes this paper and presents a brief outline of future work.

2 WS-SOA-based composite applications' shortcomings regarding user services interaction

The massive deployment of user-centric services on the internet demands services that must be accessible for all users (not only enterprise stakeholders). Therefore, services should flexibly and dynamically support common daily processes (both business processes carried out at any time by companies and processes conducted by individuals or groups in their daily life) (McAfee, 2006). Users will see the tools supporting their daily work replaced by composite applications based on web services, but traditional web

services are not tailored well enough to users and their daily processes. Obviously, a SOA, as it was originally conceived, represents an architecture focused fundamentally on a B2B context. It is weak for B2C problems, since it does not offer the best prospects for dealing with user-service interaction (Schroth and Janner, 2007). We can tackle its shortcomings from three different perspectives:

1 *SOA's aim*: SOAs merely aim at facilitating seamless machine-to-machine collaboration. SOA deployments are very abstract and invisible to users. SOA customers of choice are medium-sized or larger corporations rather than normal end users along the long tail of the internet. Therefore, with SOA, normal internet users with little IT expertise have not been able to easily retrieve and use services, because services mostly reside within company boundaries and are only accessed for professional use in a corporate context.

2 *SOA's technology*: Apart from SOA's aims, this architecture relies on a set of complex standards that are far from user friendly (OASIS, 2003). Because, technically speaking, SOA is extremely complex, there needs to be one or more expert players within the value chain to build and provide solutions for their customers. In contrast to this one-to-many value chain model of numerous SOA use cases (where one expert serves many clients), new value chains should begin to be mostly loosely coupled (many-to-many) networks of self-managed, self-sufficient users who can offer and consume web resources.

3 *SOA's government*: Finally, SOAs are subject to clearly defined regulatory frameworks since they mostly exist in the corporate context. The design, provision, maintenance and coupling of services must be compliant with legal frameworks. Therefore, they do not allow the flexibility that the described new user-services interaction model appears to need.

3 How people interact with current services on a day-to-day basis

Imagine an anonymous end user of any enterprise and her daily user-service interaction. One of the tasks that she performs as part of her job is to find out and interact with different services, resources and information, either by sharing knowledge with other parties (enterprises, knowledge workers, web intermediaries and so on) or looking for the information, services and resources herself. In the latter case, she usually spends a lot of time scrutinising UDDI web services or surfing the blogs, forums and boards of the different service websites that she or her company are acquainted with (Alonso *et al.*, 2004). This tedious task gets even worse when our end user realises that, depending on the service she wants to use – which can be provided locally at her company or by another service provider – she has to fill in application forms with several kinds of data formats to deal with completely different websites where she can either invoke web services in the application online or get the information to manage the service. She also has to take into account the different invocation processes to use the services properly and enter the information in the correct format. This end user is looking for some kind of facility that she can use to easily manage different services, resources, information

and, ultimately, back-end legacy and its related knowledge by composing an instant application from the services and back-end provided by each service provider. This would be displayed via a single user-centred face, regardless of the data/functionality source or particularities (McAfee, 2006). She should even be able to manage resources and services that have been personalised according to her tastes and also by context, to discard the knowledge she is not interested in and have interesting information rendered in the best user interface for her current access device (Dey, 2001). For the use case definition we shall define a real-life scenario: e-commerce. Of the different remote computing services offered over the internet (e-Bay, Yahoo! Shopping, *etc.*) we are going to consider Amazon Web Services (AWS). AWS is a collection of online services for other websites or client-side applications. It is accessed over HTTP using REST and SOAP protocols. To achieve our aims we shall use the marketing service Amazon E-Commerce Service (ECS). One of the ECS API's main capabilities is to provide clients with access to detailed product and pricing information, complex search capabilities, customer reviews of products, community and merchandising tools, shopping cart management and so on. ECS operations give developers access to much of the information available on Amazon's website. The operations are classified below by the type of information the applications can search and look up using ECS. Generally, any ECS request will need:

- a public access *Key Id*, which is associated with a given developer. Any service request must include that Key Id
- an *Operation Type*, which can be any of the listed operations
- *Parameters*, which will depend on the operation type, *e.g.*, an Item Search operation will need a keyword parameter and the searched product type
- (optional) a *Response Group*, a parameter that tells the service what information is to be returned. By default, the response groups are Request (the response will include information about the request) and Small (information with few details)
- (optional) an *Associate Tag* for statistical issues. This tag represents a given reseller (if the developer is a reseller, she will have a tag associated with her Key Id).

ECS is offered in several ways. On the one hand, web services are offered as usual, as a SOAP service, wrapped in the most common languages (Java, C#, PHP, *etc.*) to offer the developer a higher level of abstraction. In Java, for instance, they are offered as a library with several classes to access the services.

On the other hand, ECS is offered as a *REST-like service* (Fielding, 2000). Amazon offers a REST interface to access its services, fostering user-friendly approaches. A request would be as follows:

```
http://webservices.amazon.[com/co.uk/fr/...]/onca/xml
?Service=AWSECommerceService
&AWSAccessKeyId=[Access Key ID]
&Operation=[Operation:ItemSearch,CartCreate...]
```

The desired parameters for each operation are codified using the HTTP in a REST manner. For instance, an ItemSearch operation would be:

```
http://webservices.amazon.com/onca/xml
?Service=AWSECommerceService
&AWSAccessKeyId=[Access Key ID]
&Operation=ItemSearch
&SearchIndex=Books
&Keywords=dogs
```

The response to a given request is an XML document. This document can be formatted to any other representation using XSLT. This service is offered by the service itself, using the style parameter:

```
&Style=http://www.yourdomain.com/your-xsl-style-sheet.xsl
```

Even using this REST interface, which is more flexible, understandable and user-friendly than SOAP services, users without programming skills cannot manage to create their own self-service solutions to deal with their unique requirements and necessities. Initiatives like these offer a uniform way to manage back-end systems easily. Even so a little chunk of either application or JavaScript code in an html page or a little gadget still needs to be developed to invoke these AWS. The problem is that our end user knows about business logic in her domain, but is a nontechnical person and has no programming skills (Hgg *et al.*, 2006). As a first approximation to achieve a solution to these problems, some of these service providers could provide end users with isolated gadgets to facilitate the back-end management process. This is a rather significant enhancement with respect to the initial situation and represents the current state of the art of enterprise mashups and Web 2.0 ideas (Schroth and Janner, 2007). However, there is room for further improvement: the end user would be grateful for not having her desktop flooded with lots of gadgets that do the same thing, namely giving her separate access to some isolated service or functionality. End users, playing the role of knowledge workers (Davenport, 2005), mostly need a few (complex) gadgets that provide them with all the relevant business logic they require, assuring standardised and easy interaction with the desired services and back-end legacy (Hgg *et al.*, 2006). These (complex) gadgets will be the basic building blocks in our new approach to composite applications development, and their interconnection will be able to support real functionalities and solve real daily problems. Moreover, the resources and application gadgets of our end user's interest could be more easily published and discovered by leveraging their semantic description to exploit collaboration and the emergence of collective intelligence in her company.

The Human-Computer Interaction (HCI) will be improved and enriched in the future internet of services as one of their most relevant key features. Following this approach, many solutions related to these issues are currently proliferating, like smart conversational agents (Pilato *et al.*, 2008) or new languages for verbal and graphical interactions between users and services through chatbots (Pirrone *et al.*, 2008).

Another key feature devised in this future Internet of Services is the ubiquity of the previously mentioned HCI. Users will be able to access flexible and user-friendly services anytime, anywhere. In this line, several research projects are arising, such as Mobile Information Systems (Bohl, 2007), which would definitely affect everyday life.

4 Design principles enabling the Internet of services

The evolution of web 2.0 sites and applications is a testimony to the progress in improving user relevance and service usability (O'Reilly and Musser, 2006). However, current service front-ends are far from meeting end-user expectations (O'Reilly, 2005). Applications are still based on monolithic, inflexible and unfriendly User Interfaces (UIs). This is a serious obstacle for achieving the benefits of the Internet of Services. To build the next generation service front-end for this ecosystem of services, we propose these guiding principles:

- *end-user empowerment* – This refers to enhancing traditional user-service interaction by facilitating the selection, creation, composition, customisation, reuse and sharing of applications in a personalised operating environment (Smith, 2006).
- *seamless context-aware user-service interaction* – New-generation service front-ends should have the capability to detect, represent, manipulate and use contextual information to adapt seamlessly to each situation.
- *end-user knowledge exploitation* – This principle aims to exploit users' do main knowledge and collective intelligence to improve service front-ends (*e.g.*, tagging resources).
- *universal collaborative business ecosystems* – Enterprise systems should in corporate advanced user-centric, context-aware front-ends to enable their employees and other stakeholders to exploit and share their extensive domain expertise (North, 1990) and their business knowledge (Driver *et al.*, 2004). Employees, customers, developers and providers will collaborate to create and improve enterprise resources and processes (Anderson, 2006).

5 Considerations for a framework adhering to these principles

Now that we have reviewed the design principles of the future Internet of Services, let us look at the architecture of a specific framework based on an enterprise mashup workflow-oriented enabler that empowers its users to co-produce and share *instant applications*, *i.e.*, applications based on composition rather than programming and their building blocks. This framework has been built adhering to the design guidelines, technologies, tools and methods of two initiatives, FAST and EzWeb, as an integral solution bridging users and final services.

First of all, we should briefly review the reference architecture stack underpinning the FAST and EzWeb projects. It is clear that users must be empowered to build their own applications suited to their current situational problems. To support this, IT systems provide end users with a heterogeneous ecosystem of services.

These services use different technologies and access/invocation methods. Noteworthy too are the obstacles for dealing with user service interaction issues. Most current IT systems services, including some directly targeting end users, are not HCI friendly. In the best cases these services are the basic available resources. From this viewpoint, it is not clear how end users could build a final application adapted to their own processes. Consequently, it is important to build several abstraction layers to harmonise the exploitation of all kinds of services, enrich the services through assisted composition,

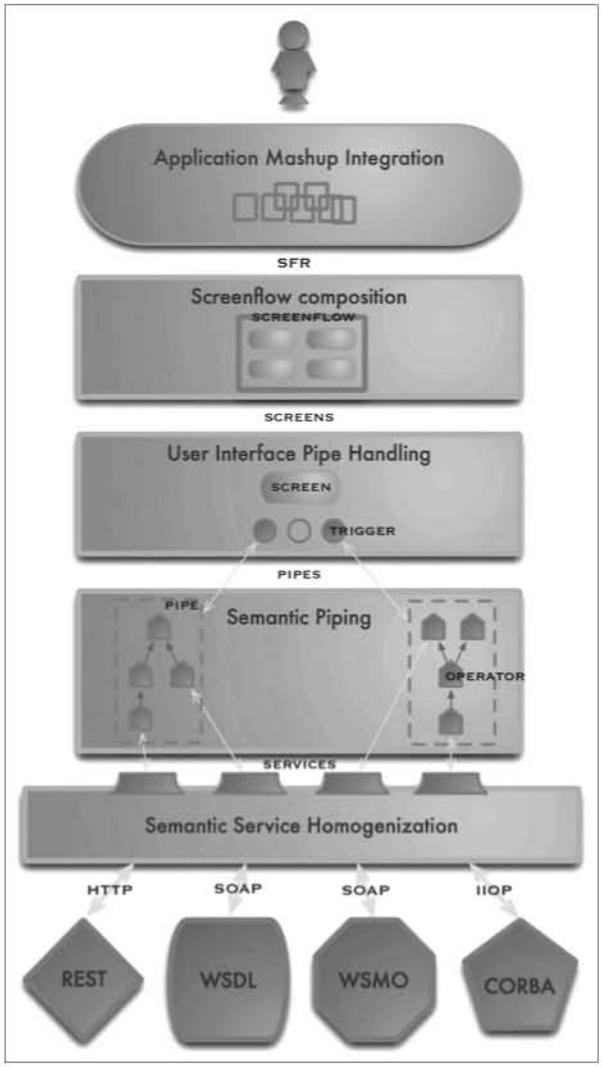
and add interactive handlers so users can use the provided functionalities directly. This way, it will be possible for most users (catching The Long Tail) to build basic service aggregations with HCI capabilities. Let us call these elements Service Front-End Resources (SFRs). Also, users can attach several SFRs to solve more complex scenarios. To do this, SFRs should provide the necessary mechanism to be properly linked. Thus, we can define four different phases within SFR development.

Figure 1 clearly identifies several layers of abstraction from the back-end services up to the end users. These elements are:

- service semantic homogenisation: Services are the basic tiles for building applications. There are different kinds of service technologies and paradigms, and there are also different solutions to enrich their interoperability and semantic information. For generality's sake, it is necessary to define a layer able to homogenise all these existing divergences, providing a general framework allowing users, and the subsequent layers, to locate, invoke and interoperate.
- with all kinds of services: *FAST gadgets can only deal with uniform resources accessed through HTTP*. Then the services must be adapted (wrapped) to be used by the gadgets. Figure 1 illustrates this wrapping as *resource adapters*. A resource (Fielding, 2000) is a simple component that owns aUniform Resource Identifier (URI) and answers the requests of the basic http verbs (*i.e.*, get, post, put and delete), thereby wrapping the access to some data or functionality (*i.e.*, above services) via a uniform interface. In the FAST gadget architecture, resources are the main components for building a gadget. They provide gadgets with both data and functionality, which will finally be shown in the gadget interface. As already mentioned, the lowest-level resources are built from the existing back-end services. Then, these back-end resources can be composed, filtered, *etc.*, by means of piping (connecting inputs and outputs using a set of operators), thereby offering the appropriate data to the higher-level layer.
- semantic piping: It is necessary to adapt, filter and transform the information retrieved from the back-end services according to the data structures and information required by the forms and interfaces. To do this, it will be possible in the semantic piping layer to define data flows between previously homogenised services and a collection of operators. Operators will manage their input data flows in a syntactic, lexical and semantic manner, and they will export a new transformed output flow. This basic mechanism allows operators to be interlinked to form highly configurable 'pipes'.
- user interface pipe handling: Pipes are associated with triggers. Enabling and activating triggers launch the execution of a definite pipe. This way, triggers are able to provide data and execute composite services from user interface events and actions. On the other hand, data obtained from triggers in this layer is bound to interface controls and presented to the users. The final result of this phase is called a screen.
- screenflow composition: The screens are interconnected, creating a nondeterministic flow of user interfaces. This screenflow generates a gadget, that is, a complex front-end screenflow resource.

- application mashup integration: finally, the gadgets are interconnected with each other to create a mashup. A mashup is an instant composite application.

Figure 1 EzWeb/FAST reference architecture (see online version for colours)



The main objective of this architecture is to allow users with no programming skills to build applications. However, this process will involve several user roles, whose profile and technical skills will depend on the tools and facilities provided to aid their job. In general, activities involving back-end access will have need of more technical expertise than others directly involving HCI issues. Tools like FAST will be in charge of helping users with service semantic homogenisation, semantic piping and UI pipe handling, while EzWeb is now a solution for the Application Mashup Integration phase.

In the following sections, we provide the technical details of both Fast and EzWeb to improve the understanding of the proposed approach.

6 Enabling top-down development of service front-ends from the gadget perspective

The main objective of FAST is to create a new visual programming environment that will facilitate the development of complex front-end gadgets, involving the execution of relatively complex business processes that rely on back-end semantic web services, *i.e.*, considering the service front-end development from the gadget perspective. Later we will change the focus and consider it from the mashup perspective.

FAST will provide technical users with the necessary tools to build complex gadgets in a visual top-down fashion (development starts from the screens and UI elements best suited to user requirements and moves down to back-end services), always taking into account the user perspective. A complex gadget manages the business processes characterised by states and state transitions, and also requiring user interaction. These tools will allow the design of the different screens with which users will interact. This way tool users will be able to define their relationships with the gadgets and with the resources and services provided by the back-end.

The approach adopted will be user – rather than program – centric. Instead of first building programs that orchestrate the available semantic web services and then trying to figure out how to implement interaction with the users at given points of the process execution flow, programmers will start from the front-end gadgets that the user will see and interact with. Then they will visually establish the connection to back-end web services, if necessary going through process execution flows. Programmers will apply an approach that looks like UML sequence diagrams to visually establish this connection.

The FAST project believes that this new user-oriented approach is feasible and that it will overcome the limitations of current business process engine approaches. In this approach, programmers will visually manage and connect front-end gadgets, screenflow resources and back-end services.

6.1 FAST platform architecture

The first point for consideration is the general architecture of the FAST platform. This is shown in Figure 2 and depicts the two main elements of FAST: the resource catalogue and the Integrated Development Environment (IDE).

The resource catalogue can be viewed as the FAST platform's back-end (note: not the back-end of a gadget). It serves several important purposes, such as storing and indexing gadgets, resources, back-end services, user profiles and histories.

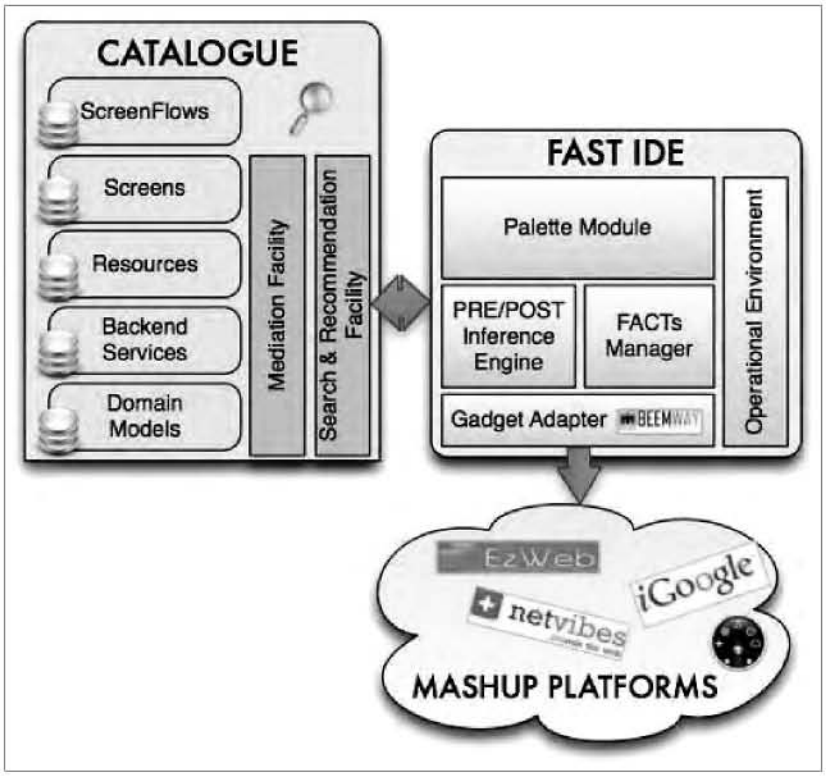
The FAST IDE is composed of several main architectural elements:

- Palette Module: Users can use this module to choose from preselected domain-specific building blocks. At this level it acts like a local catalogue from where users can drag and drop the desired components to build the gadget.
- PRE/POST Inference Engine: This module will be responsible for detecting accessible screens, possible constraint connectors between screens and produced facts. It uses an algorithm employed during the design process. This algorithm offers predictable information about the different rules inherent to the current model under development.

- FACTs Manager: This module enables navigation, inspection and selection of the desired facts inside the domain-specific knowledge model associated with a given complex gadget. It can also be used to link concepts to data, conditions or another knowledge-sensitive building blocks.
- Operational Environment: This is the module where users create their gadgets. It shows the UI giving easy access to module functionalities, simplifying the usability of the whole development process.
- Gadget Adapter: This module is responsible for exporting defined gadget behaviour to a variety of mashup platforms. The goal is to develop a functionality that allows users to adopt a develop-once-and-deploy-many strategy.

Note that the created gadgets should be compatible with most existing and future mashup platforms.

Figure 2 FAST platform architecture (see online version for colours)



6.2 FAST IDE mock-up

This section introduces an early design of the FAST User Interface (a mock-up). It will show the UI layout and its main components. As Figure 3 illustrates, there are four different areas:

1 Toolbar Area: This area includes commonly used buttons, like cut, copy, paste and other general actions. The actions do not depend on the specific task being done.

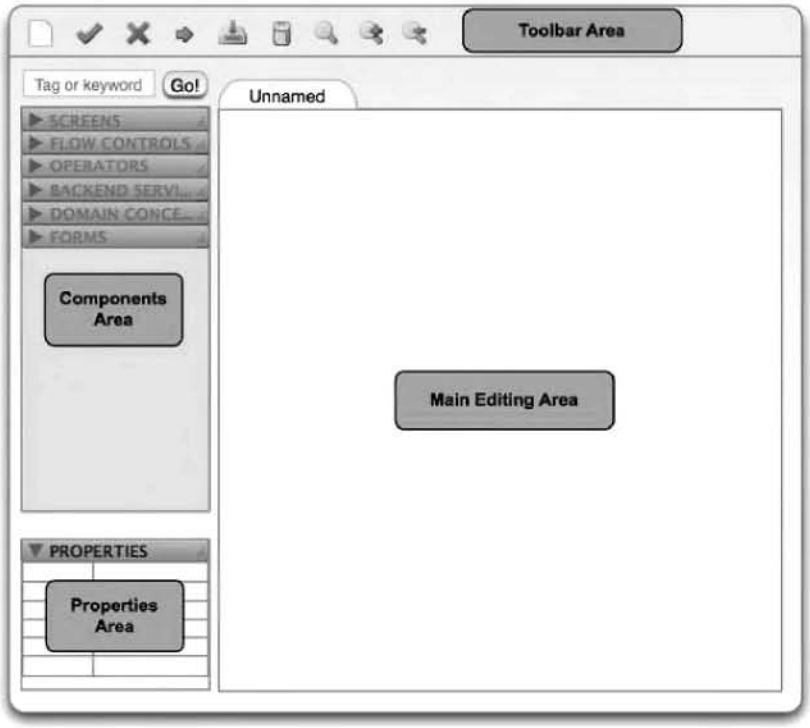
2 Main Editing Area: This tabbed region covers most of the screen and can hold artefacts involved in the gadget development process. Each tab represents one artefact that is being edited. The remaining areas will reflect tailored behaviour for the kind of artefact in use.

3 Components Area: This area provides access to a relevant subset of building blocks available in the FAST catalogue and other useful components. This subset includes the categories of building blocks needed for editing the current artefact.

4 Properties Area: This area is very similar to the property editors of the most extended development environments. It shows a grid of property-value pairs of the selected object within the artefact in use.

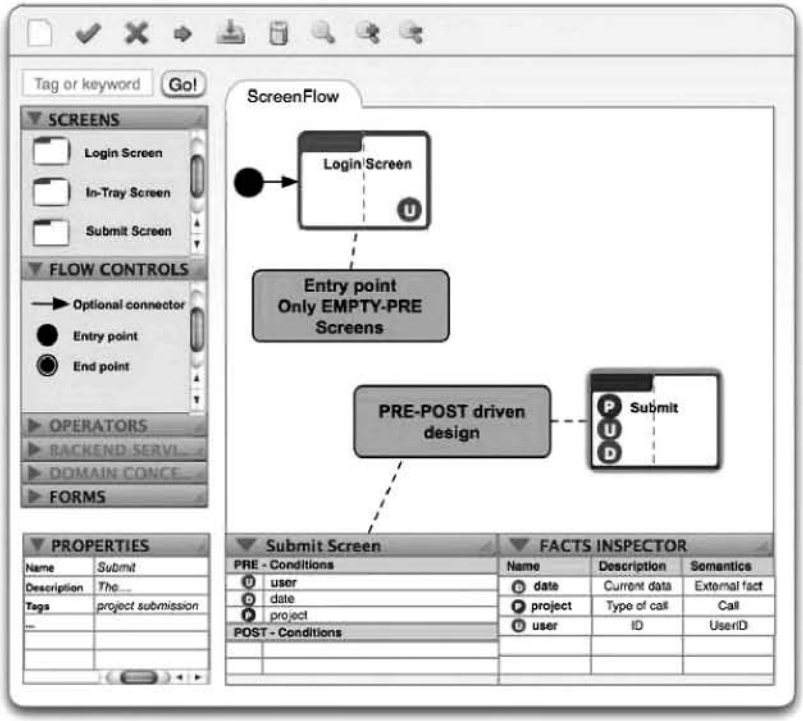
Taking the FAST approach, a domain expert will use this tool to compose a complex gadget that meets the end user's needs. To do this, the expert selects the appropriate button in the Toolbar Area to create a new screenflow.

Figure 3 General FAST application layout (see online version for colours)



Then, a new empty tab is created in the Main Editing Area, and the Components Area reflects the new situation by enabling the Screen Palette as shown in Figure 4. Also, other tabs for designing the screen and form layout appear in this area. These tabs and their functionality will be explained later in this document.

Figure 4 Screenflow tab in use (see online version for colours)



6.2.1 *Screenflow tab*

When designing a screenflow, the domain expert has to choose the screens in the screenflow. Each of these screens has a set of attached *preconditions* and *postconditions* that will be used to drive the transition between screens through a set of *produced facts* during the screenflow execution. A screen, then, has two possible states: reachable and unreachable, represented by a green or red background, respectively. If all the preconditions of a screen are fulfilled by the facts produced in the screenflow execution, the screen is reachable. Note that these screens can be found in the Screen Palette within the Components Area.

Taking into account the above, when the Main Editing Area contains the Screenflow tab, this area will be subdivided into three different areas with specific objectives and interactions:

1 Screenflow Canvas: This area shows the set of screen instances that make up the screenflow, also visualising their reachability and depicting their pre- and postconditions.

2 PRE/POST Area: When selecting a screen, this area shows the related set of pre- and postconditions.

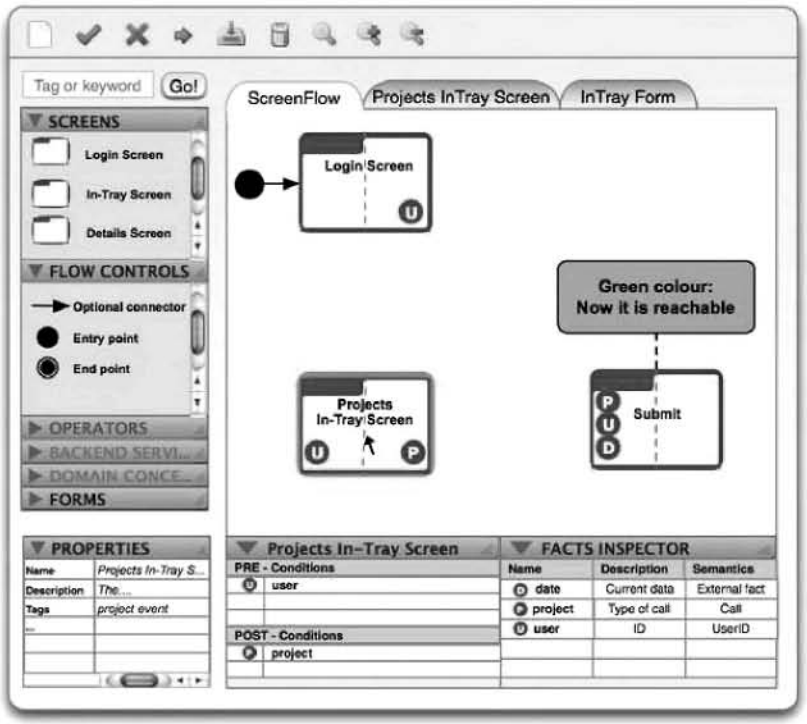
3 Facts Inspector: This area will display each available fact in the screenflow. These facts can represent both domain-specific concepts and data, and are subdivided into two types: external facts and produced facts. External facts come from outside the gadget (*e.g.*, delivery context, user input) and produced facts all come from the postconditions of the available screens. For each fact, the inspector shows details such as name, type (referencing an ontology), textual description and so on.

As Figure 4 shows, the domain expert adds a screen (using a drag-and-drop action) called submit. This screen is colored red since it is currently unreachable. To make it reachable, the screen needs two facts, U and P. The fact inspector describes U and P as user and project facts, respectively. Consequently, the domain expert uses the palette to browse the FAST Catalogue looking for screens whose postconditions can satisfy the submit screen preconditions. The FAST Catalogue could use semantics to simplify this process and recommend the most appropriate screens.

To fulfil the user precondition, the expert can easily find a login screen and drop it in the canvas. Note that, at this point, the submit screen is still unreachable, and the expert looks for a screen to satisfy the remaining precondition.

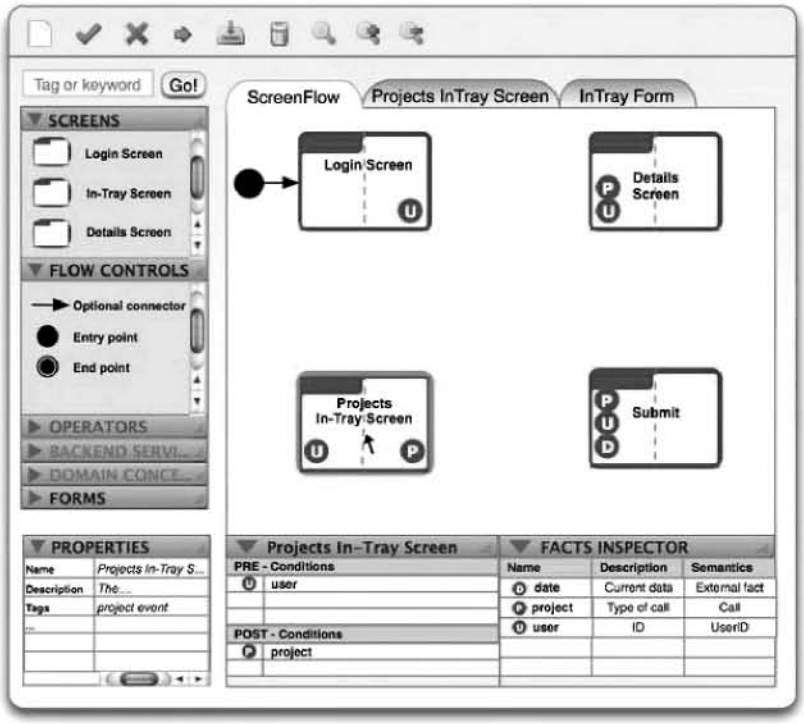
Then the user finds a predefined project in-tray screen, also dropping it into the canvas. Now, the submit screen turns green (Figure 5), and the screenflow is ready for execution (all the screens are reachable). Although the gadget is executable, the domain expert decides to add an entry point component to the login screen, establishing it as the first screen to be shown when the gadget executes.

Figure 5 Facts verification and management in the screenflow (see online version for colours)



Due to the PRE/POST mechanism, there would be no obstacle to adding a screen whose preconditions were satisfied by the current facts present in the screenflow. It would also be possible to have screens that do not have any postconditions. This is illustrated in Figure 6, where the domain expert has added a details screen.

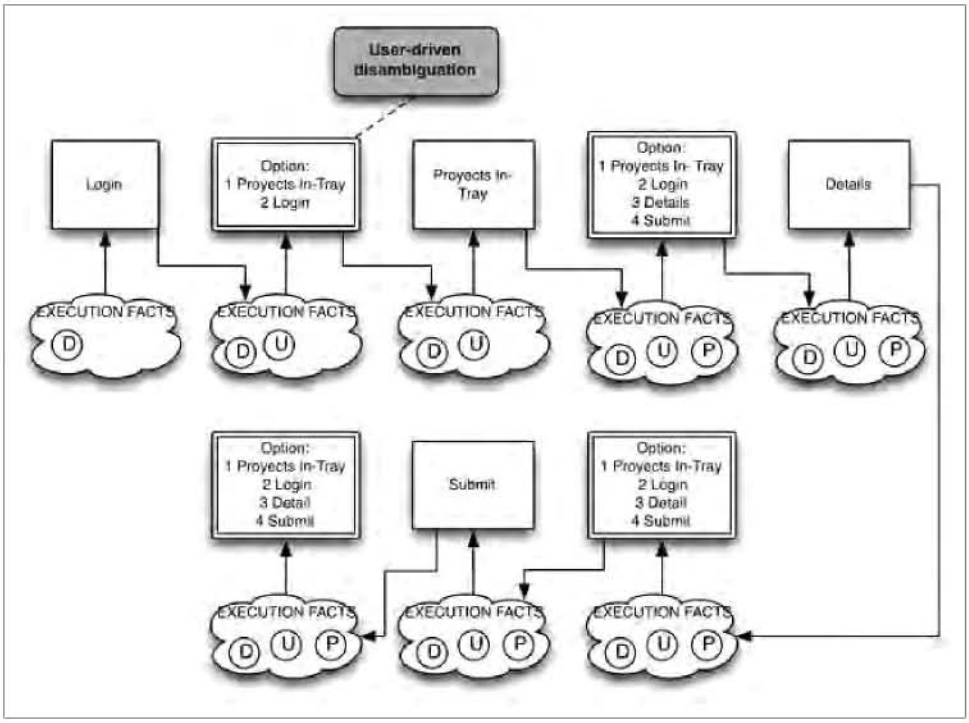
Figure 6 Addition of screens without postconditions (see online version for colours)



Note that, for the time being, the domain expert has not explicitly defined any transition between screens. However, the FAST tool has already done this for her implicitly by means of the PRE/POST and fact mechanisms. This is explained in the following execution example. Figure 7 shows the execution flow of the created gadget. This flow follows these simple rules:

- It is possible to show only screens whose preconditions are fulfilled at the time of transition. These screens are called enabled screens.
- If there are two or more enabled screens, the conflict is solved by means of a menu from which the user can choose the next screen. If there is only one possible transition, it will be done automatically.
- After screen execution, the execution facts set are updated with the output postcondition. Consequently, the set of enabled screens could change.

Figure 7 Nondeterministic execution flow (see online version for colours)



The above rules guarantee the transition between screens. This is the reason why the domain expert did not need to define them. But there are some scenarios in which the transition between screens must be set explicitly. FAST provides domain experts with a set of *flow connectors* that they can use to define the specific flow and improve the user experience:

- Init connector: It sets the first screen to be shown in the execution. It cannot be associated with a screen that has a precondition.
- Option connector: This connector explicitly fixes the possible transition destinations, creating a subset of enabled screens. If there is only one possible transition, it will be made automatically.
- Advanced option connector: This is an option connector that only makes sense when a logical constraint is satisfied.

Using these connectors, the domain expert decides to establish a predefined flow, as shown in Figure 8. In this scenario, the gadget execution would be as illustrated in Figure 9.

Figure 8 Forced execution-flow order (see online version for colours)

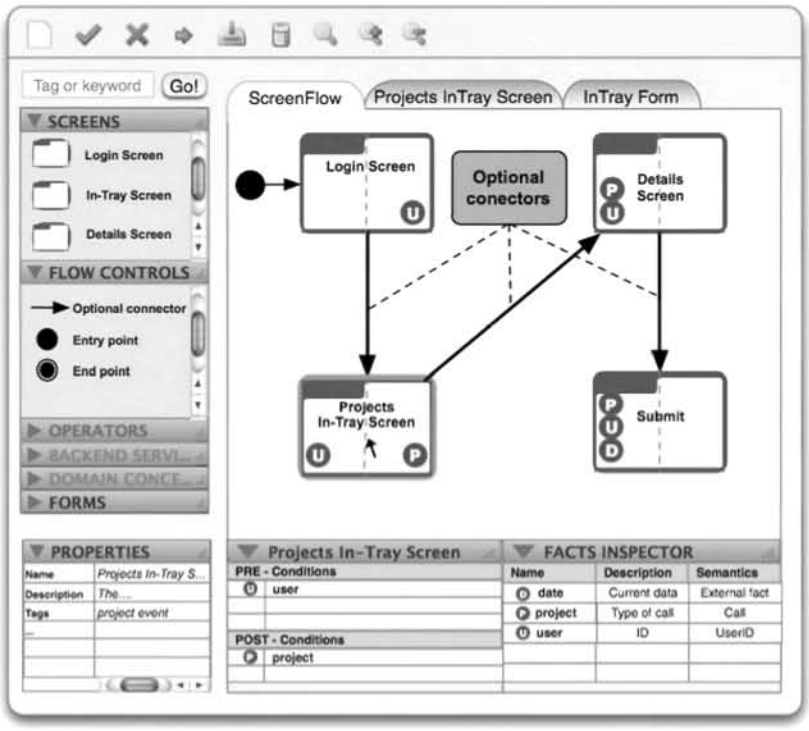
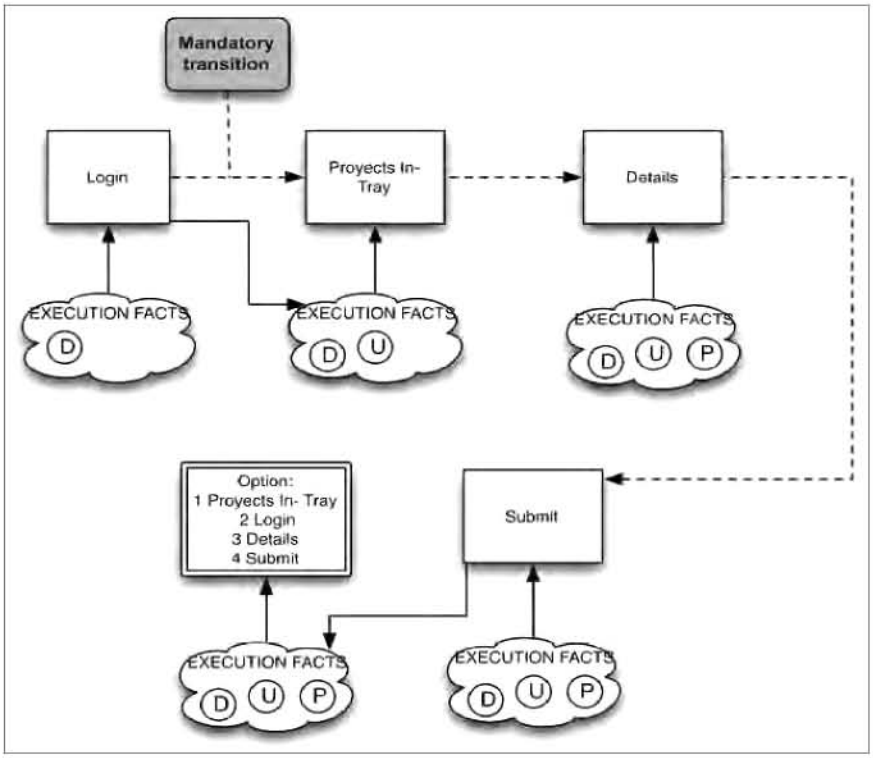


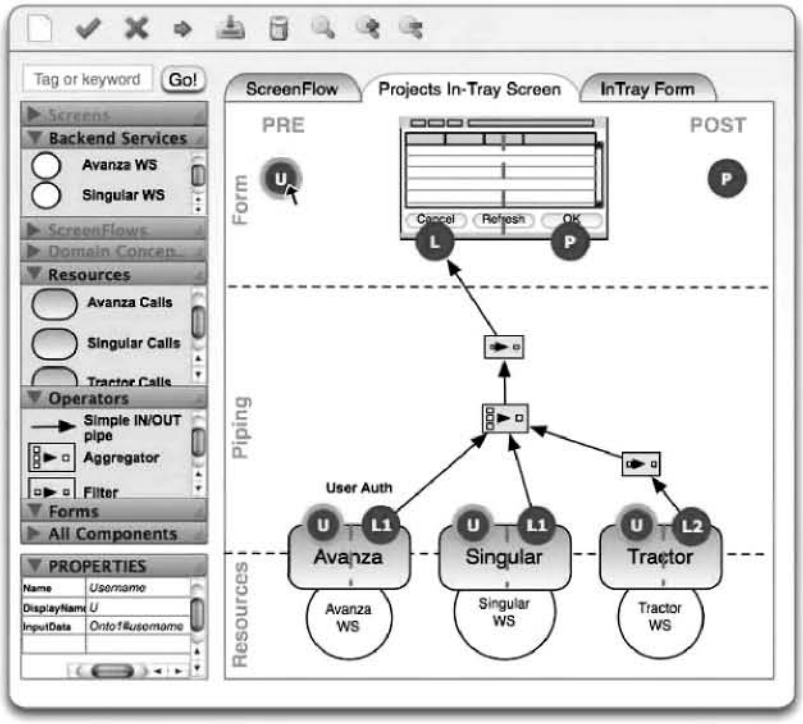
Figure 9 Resulting execution order (see online version for colours)



6.2.2 *Screen tab*

The screen tab has three separate spaces that match up with different component tiers. This builds a screen tab (Figure 10). The top layer contains an icon representing the form on which the screen is based. This icon has as many placeholders as the underlying form has preconditions and postconditions.

Figure 10 Screen tab (see online version for colours)



The bottom layer represents the actual web services that can be dragged and dropped from the palette (white circles), as well as their available interfaces (REST-ful, SOAP, *etc.*). These interfaces are represented by blue boxes in Figure 10. Also, the service inputs and outputs are modelled as preconditions and postconditions.

The remaining area between them is the operator area. Simple and complex operators can be composed to filter, aggregate, mix or adapt the services that are finally connected to the form placeholder. Despite this possibility, simple screens can be associated directly, as in our example.

At the end of the day, all the web services preconditions must be met by service postconditions, screen preconditions or operator postconditions, whereas not all the form preconditions have to be satisfied. Unfulfilled form preconditions will become screen preconditions.

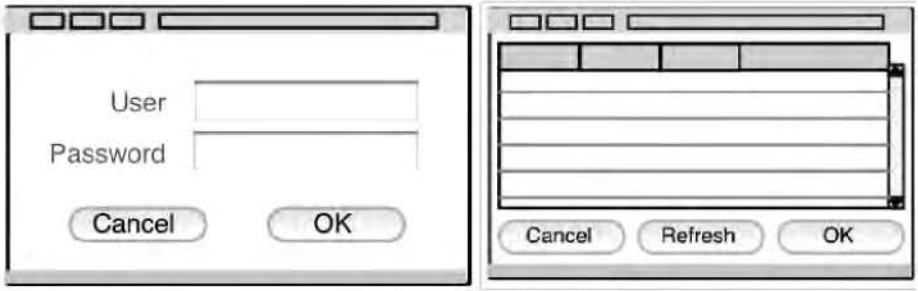
6.2.3 *Form tab*

Form characteristics determine the screen tab configuration options. Therefore, a detailed study of these characteristics is required to fill in the whole picture. A form has several characteristics:

- form preconditions and postconditions. These can be seen as hooks for connecting a service and other stuff that meet certain constraints.
- properties. These are a set of variables for customising the form. This makes the forms general enough to build a wide range of gadgets from just a few customisable forms.
- events. An event is a trigger for actions. The selection of an item in the event tray is a good example of an event.

The login form and the event tray form are examples of forms and their representations can be shown in Figure 11.

Figure 11 Login and in-tray forms (see online version for colours)



6.3 *Runtime production system in FAST*

A gadget execution is a sequence of FAST screens interacting with the user, which is user driven. However, the internal gadget state (knowledge base) and the restrictions associated with those screens determine which transitions are eligible at any time.

FAST screens are to be construed as production system rules. Screen preconditions act as the rule antecedent, that is, whenever they are all satisfied, the rule is enabled. When the production system triggers one of the enabled rules, the respective screen is shown and the control is transferred to the user. After the interaction and service invocations for which the screen is designed, a set of postconditions are added to the knowledge base. Therefore, postconditions can be seen as the consequent rule. This section explains the above FAST runtime production system.

6.3.1 *Facts and the knowledge base*

The knowledge base represents the gadget state by means of a set of individual facts. A fact is just an instance of an ontological concept identified by a URI. For the following mock-up iteration, the knowledge base only admits one instance per class in order to simplify the system.

Ontologies define classes of concepts and descriptive properties. Consequently, a fact description includes the class resource identifier (URI) and the set of properties defined in the ontology. Ontologies are needed at design time to generate the knowledge base data structures.

At the end of the day, a fact is a data structure keeping the value of those properties and stored in the knowledge base as an instance of a domain concept. The knowledge base is a sort of sophisticated hash map enabling facts identified by URIs to be inserted, retrieved and removed. Once the fact is retrieved, it is possible to manipulate its properties.

For instance, the ontological data of a hypothetical domain can be compiled in an RDFS document as follows:

```
<?xml version="1.0"?>
 <DOCTYPE rdf:RDF [
 <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
 <!ENTITY rdfs 'http://www.w3.org/TR/1999/PR-rdf-schema-19990303#'>
 <!ENTITY xsd 'http://www.w3.org/2001/XMLSchema#'>
 ]>
 <rdf:RDF xmlns:rdf="&rdf;"
 xmlns:xsd="&xsd;">
 <rdfs:Class rdfs:about="http://blibli.bla/bla#item" rdfs:label="Item"/>
 <rdfs:Class rdfs:about="http://blibli.bla/bla#book" rdfs:label="Book">
 <rdfs:SubclassOf rdf:resource="http://blibli.bla/bla#item"/>
 </rdfs:Class>
 <rdfs:Property rdfs:about="http://blibli.bla/bla#asin" rdfs:label="ASIN">
 <rdfs:Domain rdf:resource="http://blibli.bla/bla#item"/>
 <rdfs:Range rdf:resource="&xsd;string"/>
 </rdfs:Property>
 <rdfs:Property rdfs:about="http://blibli.bla/bla#price" rdfs:label="Price">
 <rdfs:Domain rdf:resource="http://blibli.bla/bla#item"/>
 <rdfs:Range rdf:resource="&xsd;decimal"/>
 </rdfs:Property>
 <rdfs:Property rdfs:about="http://blibli.bla/bla#isbn" rdfs:label="ISBN">
 <rdfs:Domain rdf:resource="http://blibli.bla/bla#book"/>
 <rdfs:Range rdf:resource="&xsd;string"/>
 </rdfs:Property>
 </rdf:RDF>
```

Two classes are defined: item and book. The book is a subclass of the item, extending its definition. Properties have a domain (subset of classes to which the property is applied) and a range. The range is another class defining the property data type. It is possible to use RDF and XML Schema data types that provide basic types and complex structures. Note that RDF properties can be inherited and specialised.

The data structure to be used during gadget execution can be extracted from that RDF information. For instance, a snapshot of the runtime knowledge base containing a book and item instances can be represented as the following JSON structure:

```
{
 "http://blibli.bla/bla#book" : {
 "http://blibli.bla/bla#asin" : "8420658804",
 "http://blibli.bla/bla#price" : 123.12,
 "http://blibli.bla/bla#isbn" : "978-8420658803"
 }
 "http://blibli.bla/bla#item" : {
 "http://blibli.bla/bla#asin" : "B000JZ9ATS",
 "http://blibli.bla/bla#price" : 12
 }
 }
```

6.3.2 Screens as rules

As explained earlier, rules are derived from screens. Therefore, FAST screen metadata must include a formal definition of pre- and postconditions.

A precondition is defined as a conjunction of positive or negative atoms. Each atom is a concept URI plus restrictions (zero or more logical expressions) on concept attributes represented by URIs. Therefore atoms express a restriction on the knowledge base and can require the existence or absence of a concept.

The precondition has two purposes during gadget runtime: screen enabling and fact selection. When the production system is invoked, all the preconditions are matched against the knowledge base in order to consider the enabled screens. Once the next screen is established within the enabled screens, the facts that matched the preconditions are passed as input data.

An example of a precondition using URIs as identifiers, namespaces and an abstract syntax is:

```
foaf:user /\ not amazon:promotional /\ amazon:item (amazon:price < 10)
```

This condition establishes that a user fact must exist, no promotion must exist and the selected item must have a price of less than 10 euros.

In contrast, postcondition information is more useful during the design than during gadget execution. Postconditions codify the facts that are going to be deleted, added or updated in the knowledge base as a result of the execution of the screen. Sometimes, one screen can drive different facts depending on the user interaction and the back-end services. In such cases, more than one postcondition can be declared. Briefly, a postcondition is a conjunction of positive or negative atoms, and these atoms are a concept URI. Consequently, postconditions are simpler than preconditions as shown in the following abstract syntax postcondition example:

```
amazon: item /\ not amazon promotional
```

This means that a new item fact will be added to the knowledge base, and the promotional code fact will be removed. Any preexistent item will be overwritten by the new one.

The relevant sections of the screen metadata representation must follow the same structure. A combination of the above two examples in an XML document looks like this.

```
<fast:screen ...>
 ...
 <fast:pre>
 <fast:atom fast:sign="positive" rdf:resource="http://...#user"/>
 <fast:atom fast:sign="negative" rdf:resource="http://...#Promotional"/>
 <fast:atom fast:sign="positive"
 rdf:resource="http://...#Item">
 <fast:restriction rdf:resource="http://..#Price">
 price < 10
 </fast:restriction>
 </fast:atom>
 </fast:pre>
 <fast:post>
 <fast:atom fast:sign="positive" rdf:resource="http://...#Item"/>
 <fast:atom fast:sign="negative" rdf:resource="http://...#Promotional"/>
 </fast:post>
 ...
</fast:screen>
```

6.3.3 Variables

Quite often, different atoms within a screen precondition should be linked together in order to express some relationships. For example, the knowledge base can contain a user fact and a selected item fact. Only if the user is the item owner can a hypothetical ManageItem Screen be triggered. The ManageItem precondition can be written using variables as follows:

```
foaf:user (foaf:username = ?u) /\ eg:item (eg:owner = ?u)
```

6.3.4 Production system dynamics

The screen representation during gadget execution is a data structure with few differences with respect to the XML representation. Due to performance restrictions, XML parsing should be subject to strict constraints at gadget runtime. A more efficient technique is to send JSON structures from the server side.

```
screen = {
 uri : "http://.../",
 PRE : [
 { concepts : [
 { concept : "http://.../",
```

```
positive : true },
{ concept : "http://.../",
positive : true }
],
constraint : function () { ... }
}
],
POST : [ // disjunction
[ // conjunction
{ positive : true,
concept : "http://.../"
}
]
]
form : "uri...",
piping : "..."
}
```

One important point is that preconditions have embedded functions that are able to check the restrictions shown in the XML screen description rather than actual restrictions. As a result, the difficult bits of the process can be simplified on the server side (*e.g.*, variable matching).

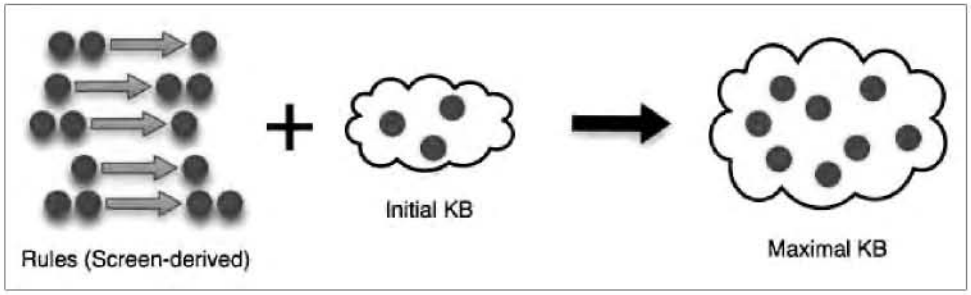
6.4 Design time production system in FAST

Design time requirements are nothing like runtime requirements. On the one hand, screen and fact reachability should be estimated and used as graphical feedback. On the other hand, smart screen recommendations should exploit the available semantic information about the screens and goals in order to best help the user.

6.4.1 Screen reachability

Ideally, the FAST development tool should be able to determine what facts and screens could be reached during the gadget execution. This is equivalent to determining whether the knowledge base can reach a state that satisfies all the required restrictions for every screen precondition. Unfortunately, this is quite ambitious because of the memory and space a greedy approach requires and the difficulties of implementing an analytic solution.

Figure 12 Maximal knowledge base (see online version for colours)



The proposed approach is to ignore the semantic description of the screen behaviour and the fact removals. The resulting simplified rules can be executed iteratively until no new facts are added to an estimated maximal knowledge base. The screens that are enabled for the maximal knowledge base are coloured green.

6.4.2 Screen recommendation and goal-driven design

Screen recommendation and goal-driven design can be addressed by means of the same strategy. First of all, it must be possible to describe gadget goals in an easy and maybe implicit fashion. To do so, we rely on two sources of goal information:

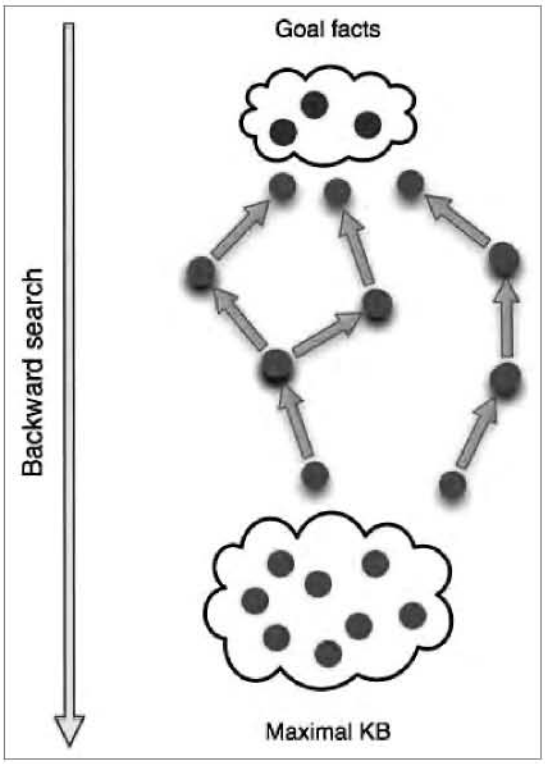
1 screen facts that are not yet reachable – If the user has dropped a screen in the gadget, it is because he/she wants that screen to be a useful part of the screenflow.

2 hand-made goal facts – If the user has a clear idea about his/her objective, he/she should be able to add a goal fact that will guide him/her during gadget design. For example, he/she can add a goal fact using a registered user concept from the Amazon ontology. Then the system will recommend screens related to that goal.

Once the goal set has been defined, the screen recommendation can be based on a backward search to try to get a path, described as a sequence of rules, from the goals to the current maximal knowledge base. Building this search graph can be time and space consuming. This is not a big problem, though, because it can be done in the background, giving the user the best partial solution at any time.

Some heuristics can be added to the recommendation system to give preference to the shortest paths and the screens used by most of the paths.

Figure 13 Goal-driven approach (see online version for colours)



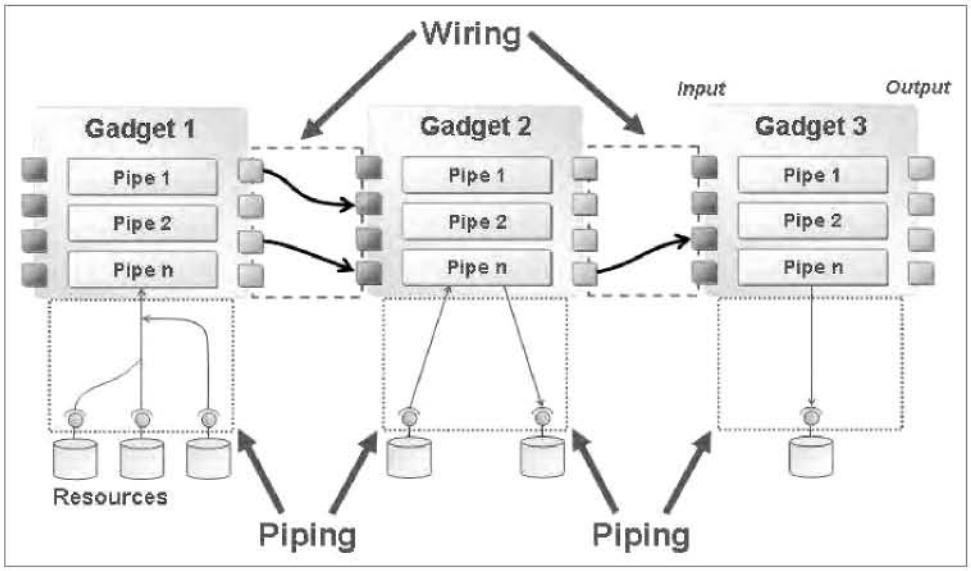
7 Enabling top-down development of service front-ends from the mashup perspective

After presenting the technical details of FAST and how it can be used to create gadgets, this section goes a step further and explains how EzWeb can be used to mash up even the above complex gadgets and other heterogeneous resources.

The way in which building blocks (that are combined to create composite applications) are related to other blocks and to their own platform is vital in a mashup platform, like EzWeb (Lizcano *et al.*, 2008). As many use cases have illustrated, these resources must intercommunicate to enable really powerful compositions and mashups. Also, they should provide a flexible and simple support for managing fixes, fits and configurations of their own look, appearance or operating modes (Smith, 2006).

From an abstract point of view, EzWeb's central driver, designed to address The Long Tail of user needs, is the lightweight resource composition style. In this style, building blocks from different contexts are reused to build individual enterprise applications. As illustrated in Figure 14, the composition takes place both in the resource layer (piping) and in the gadget layer (wiring).

Figure 14 Resource and gadget compositions building composite applications (see online version for colours)



The piping composition integrates a number of heterogeneous web-based resources defining composed processing data chains/graphs concatenating successive resources. In the gadget layer, the end user is able to wire existing gadgets together with behaviour and data relationships by visually interconnecting their input and output parameters. Users then can interconnect existing resources with each other to create increasingly complex web services and their APIs.

The next section provides the technical details of both piping and wiring compositions. It explains how EzWeb supports these issues, fostering the remixing/ fitting of resources through a simple representation of resources called template and empowering gadget-to-gadget and gadget-to-platform communication based on a programmatic idea called *metavariables*.

7.1 Template-based interoperability provided by EzWeb to its runtime components

In our architecture a gadget code is divided into two parts: a resource and an XML template. This template will allow this component to be added to each mashup platform as a new fixable/mixable building block. Templates can also be created with the aid of FAST. FAST will recommend intercommunication channels and possibilities, based on the nonsatisfied gadget pre- and postconditions and functionality, internal facts, managed concepts and so on.

Templates gather all the implementation details that have nothing in common with the resource business layer, reducing the components code coupling. Compared with the traditional object-oriented programming model called Model-View-Controller (MVC), this template part could be considered as a mix of the view-controller data set, whereas the resource body represents the model functionality. This separation is necessary to increase component creation productivity, improve performance, and reduce changes and chaos in the event of resource modifications or hacking. Since resources must, as a part of the mashup philosophy included in the revolutionary Web 2.0 paradigm, support changes, constant beta evolution and lightweight programming ideas (O'Reilly and Musser, 2006), the template is vital for component remixing, fitting, hacking and changing support.

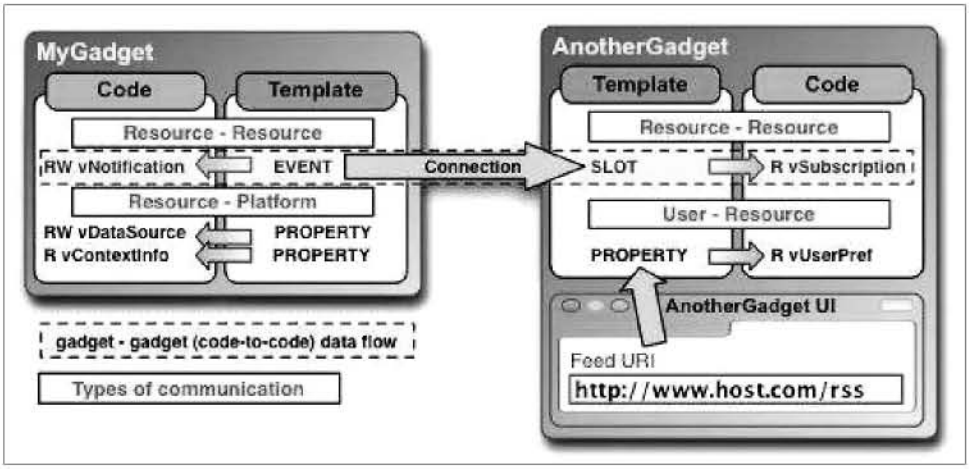
Accordingly, templates contain nothing about how resources carry out operations or calculations, encapsulating only in/out information, how to feed and extract data to/from internal processes, and information about resource visualisation and configuration (see Figure 15). Considering all these design principles, templates should include details about:

- user preferences, which condition how resources work and what they look like
- persistent properties (like context information) governing resource behaviour; these properties can be modified and read by the components' code, which interacts internally with the template
- intercommunication resource data with information about how a building block interacts with the platform and with other components. Specifically, this data is presented as events and slots. Events are propagated by the resource and can be consumed by other resources or by the platform. Slots receive events and can be consumed by the resource, that is, there are rules and codes that implement how a component should react and operate with them. These intercommunication-related issues will be explained in detail in the next section on *metavariables*.

Briefly, metavariables solve all current issues about resource communication in a smart and homogeneous way. From the resource viewpoint, a new CONTEXT information item, new data incoming through a SLOT or changes made by end users

using PREFERENCEs are tackled in the same way, invoking a callback function of R variables. This way, these events are transparent to resources and are managed automatically without further code support having to be provided.

Figure 15 EzWeb interoperability framework (see online version for colours)



Figures 16 and 17 depict a simple scenario where several gadgets are put together in a workspace as a composite application that provides end users with tourist information about trips. A Yahoo! Trip Search gadget allows users to search trips. When a trip is selected, its information modifies the whole mashup: a map shows the trip directions, an Amazon gadget lists books and other products related to the trip, and finally Flickr shows pictures of the destination. The mashup connects four gadgets using several channels of intercommunication. The entry point is the Yahoo! Trip Search gadget that generates events of the type direction (consumed by Google maps) and events of the type keyword (received by Amazon and the Flickr gadget). Figure 16 shows a complete EzWeb reference architecture, as well as the creation of the intercommunication channel. Figure 17 depicts the resulting mashup.

Figure 16 EzWeb wiring management (see online version for colours)

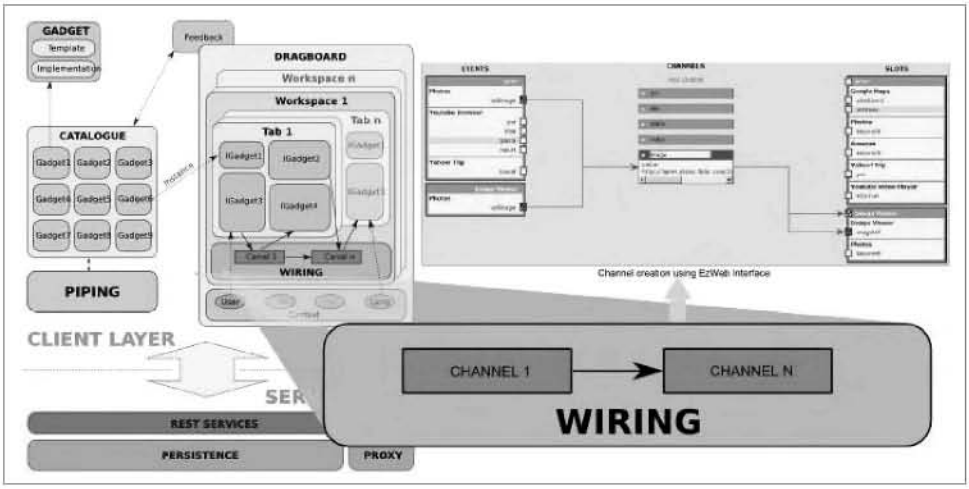


Figure 17 Final mashup and New York trip search (see online version for colours)

8 Interacting with services through EZWEB/FAST-based front-ends: proof of concept

As a proof of concept, this section shows the application of our framework (based on the FAST tool and EzWeb mashup platform) to the domain problem presented in Section 3. This scenario is only one example of the many solutions that could be developed based on the proposed, novel user-centric SOA-oriented framework. This section explains a composite application deployed on an existing prototype of the FAST and EzWeb platforms. A service-oriented environment is created by visually attaching different resources and complex gadgets to each other and to the enterprise back-end.

In the use case, we need to manage screens based on AWS ECS through its REST interface. From these capabilities we will develop a collection of *ad hoc* screens, available through the screen palette, that will provide FAST users with e-commerce functionalities. The above description of the screens includes which services to invoke and a description of each screen. All these screens therefore wrap REST services following a visual drag-and-drop-based composition and add a user interface thanks to the FAST tool:

1 *Product Search.* This screen enables users to fill in a product search form that defines some filter parameters. This screen will have no contact with ECS. It will only produce an output fact, called *Filter Search Criteria* (F from now on), including all the information about the searches to be run: search index, keywords, *etc.*, defined as URI parameters (which will fulfil the REST interface).

- Output Facts (POST): **F**

2 *Product List.* This screen shows products for a given search index or combination of search indexes, according to filter criteria. The screen will connect to the server side and access the *ItemSearch* operation, passing the data received in the F fact. The server side will return the list of items that fulfil the search criteria and the client side will build a list with all the items. Clicking on an item will produce an output fact called *Item* (I from now on), including Item identifiers (ASIN, UPC and/or wish list ID).

- Input Facts (PRE): **F**
- Output Facts (POST): **I**

3 *Product Lookup.* This screen enables users to retrieve product information, customer reviews, pricing, availability, images, *etc.* The screen needs to access the *ItemLookup* operation, passing the Item identifier (usually an ASIN or UPC). Therefore, the screen needs the U fact. Additionally, the screen will offer the option of adding the current Item to the shopping cart. In this case, it will need to access the *CartAdd* operation (or *CartCreate* if the cart has not been created yet). The CartAdd operation needs a Cart ID and an HMAC Cart token (which is produced by the CartCreate operation). This information will then be stored in the U fact when the cart is created for further access to cart operations.

- Input Facts (PRE): **U**, **I**
- Output Facts (POST): **N**

4 *Price Comparison.* This screen shows a comparative list of product prices for an item. The screen will receive an I fact, and access *ItemLookup*, passing the ResponseGroups *Offers*, *VariationSummary* and *Similarities* as parameters. The user interface will offer the option of selecting one of the products in the comparison, thus producing a new I fact.

- Input Facts (PRE): **U**, **I**
- Output Facts (POST): **I'**

5 *Graphical Price Comparison.* This screen shows a graphical representation of product prices for an item. This is the same screen as above, but offers a visual rendering of the comparison.

6 *Suggested Product List.* This screen shows a list of suggested products related to an item. This screen receives an I fact and displays similar products for the user. To do this, it has to call *SimilarityLookup* with a *Medium* Response Group (to get the

products' ASIN number). This service returns the list of similar products, with their identifiers and the product name. Then the screen will show a list of products from which the user can choose one product, thus producing a new I fact.

- Input Facts (PRE): **U**, **I**
- Output Facts (POST): **I'**

7 *Shopping Cart*. This screen is useful for managing a shopping cart of products that a user wishes to purchase. The main shopping cart actions that a customer can perform include: get cart contents, modify cart entries, clear all items from a cart and complete the purchase. To do this, it will have to access Cart operations *(CartClear, CartGet* and *CartModify)*. Cart operations always return a purchasing URL. Therefore, completing the purchase involves creating a new fact called *Purchase* (**P**). This fact will contain the purchasing URL only.

- Input Facts (PRE): **U**
- Output Facts (POST): **P**

8 *Purchase Lookup*. This screen shows the Amazon-supplied URL to complete the purchase. The screen will offer the user the option of purchasing the existing shopping cart. As mentioned above, Amazon ECS does not offer purchasing services; this is done at the Amazon site. Consequently, this screen will access cartGet to show to the user brief information about the shopping cart only. It will also offer the user a link to access the Amazon purchasing page.

- Input Facts (PRE): **U**, **P**.

From the set of screens, we shall define a collection of scenarios to build complex gadgets like an item browser, a product buyer, a gadget for shopping cart management or an item comparison. Figure 18 illustrates the creation of an item list of Amazon's products from the REST Amazon services to the final form.

By creating the other scenarios (developed along the same line as illustrated above), a complex gadgets can be created to search Amazon products, view product descriptions and even purchase products. The final look of this complex gadget is shown in Figure 19.

In conclusion, using FAST, even users without programming skills can create their own gadgets to deal with their own daily problems. Furthermore, the use of this FAST-tailored complex gadget in the EzWeb mashup platform, which fosters resource intercommunication and enhances flexibility in the service front-end paradigm, achieves a real open innovation (Soriano *et al.*, 2007), exploiting the created Amazon gadgets jointly with other gadgets (for example, Google maps, Flickr, or Yahoo! gadgets). Such a complex gadget is the Amazon gadget in the EzWeb workspace shown in Figure 17. This gadget displays products related to trips that have been searched by another gadget (in this case, the Yahoo! Trip Search gadget).

Figure 18 Item list gadget development (see online version for colours)

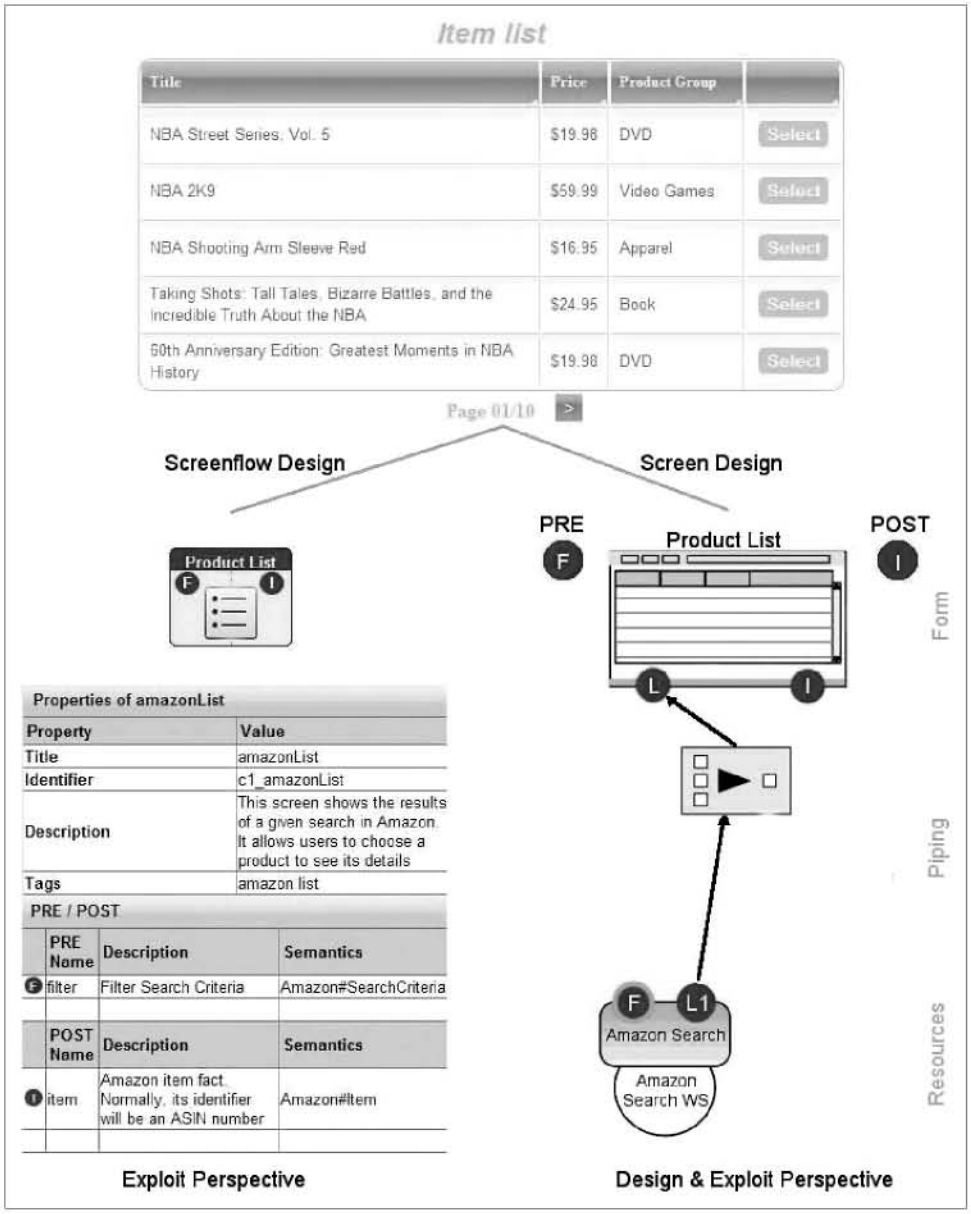


Figure 19 Final Complex Amazon Gadget (see online version for colours)

9 Contribution to the state of the art

Current mashup solutions allow users to create their own composite interface using multiple and disparate gadgets (iGoogle, Net Vibes, *etc.*) or by mixing several data sources to create another one (Yahoo! Pipes). But the use of the components of almost all of these solutions is confined to platform boundaries (*i.e.*, there is no room for interoperability with other applications: it is impossible to add an iGoogle gadget to the Net Vibes homepage). Additionally, a gadget within a given platform cannot communicate with other platform elements. These gadgets are isolated applications put together in the same window, and they cannot, therefore, be considered as composite applications. So, it is almost impossible to build real and complex applications, which need different gadgets to share data and functionalities. The currently most successful gadget and data mashup solutions are Yahoo! Pipes and iGoogle.

Yahoo! Pipes is a composition tool based on a visual web portal for aggregating, manipulating and mashing up content from around the web. The compositions are built using pipes, that is, simple commands and operators that can be combined together to create output that meets users' needs.

Therefore, this solution is oriented to data mashup, since it allows users to remix and manage contents in html, RSS feeds or XML to obtain a final data output. But they cannot remix gadgets, visual elements or interfaces to produce a new complex user interface (Anderson, 2007).

On the other hand, iGoogle is a web portal where users can create their personalised Google page. This page can mash up visual elements like calendars, calculators, news, games, to-do lists, maps, photos, weather and stuff from across the web on the user's home page. Users can add, manage, move or delete visual elements called gadgets in this dashboard by a simple search of a gadget catalogue. This portal page is used as the entry

point of the end user's experience of the entire web. However, each visual element is like a separate component, a gadget in solitary confinement, and there is no provision for gadget intercommunication.

The main difficulties a developer has to deal with to build gadget-based applications using the current approaches and state-of-the-art solutions are as follows:

- Even though there are a few platforms in which it is possible to include some hacks to achieve some sort of gadget communication, most of the mashup platforms do not support unexpected gadget intercommunication. There is another barrier to sending and receiving data between gadgets: shared data types. Ontologies (Roman, 2005) that include the different concepts (data types) to be shared by gadgets are required.
- Gadgets are considered as the new face of SOA (Hierro *et al.*, 2008) and, if they are going to be used to create complex applications, they must access and interact with remote data resources. Current data sources either do not have a friendly interface, meaning that they cannot be used easily, or do not have a standardised uniform access. To be used, they have to be hardcoded into the gadget itself.
- Another key principle of SOAs is to have a shared resource catalogue, like UDDI (OASIS, 2003), to facilitate service composition and reusability. Currently, there is no centralised shared gadget catalogue enabling users to create complex items based on simpler ones (following the mashup phenomenon (O'Reilly, 2005)).
- There is no gadget standard that includes all the requirements for building composite applications. Every mashup platform has its own gadget model, and there have been only a few attempts to create gadget standards (*e.g.*, UWA (NetVibes, 2007) tries to create platform-independent gadgets). None of them considers the complexity of a composite application made up of a number of gadgets sharing data, using and modifying remote data sources or services, and interacting with other gadgets.

Thanks to their reference architecture and by materialising the guiding principles proclaimed by web 2.0 and its application to the enterprise, the approach based on EzWeb and FAST explained in this paper tackles all these problems and handicaps.

10 Related work

Companies are beginning to focus on people as the entry point to SOA and composite applications (Coveyduc *et al.*, 2006). Thus they need a means to bridge the gap between people and services (IBM Developer Works, 2006). This is when they come up against traditional composite applications' shortcomings. Consequently, a number of user-centric composite application frameworks are beginning to proliferate.

Several European projects, ranging from the European Union Framework Programme (EU FP) to national or industry-funded projects, are addressing relevant research challenges in the area of service front-ends. They deal with research topics such as context modelling and management, or the evolution of web technologies enabling users organised into communities to mash up, configure, connect and share services in a knowledge-aware manner. Furthermore, the 2009–2010 FP7 ICT Work Programme is

to include a dedicated research topic on service front-ends in the area of service architecture and platforms for the future internet. This programme will focus on technologies enabling communities of networked users with different levels of expertise to search for, compose, configure, share and use services while supporting device- and context-aware service adaptations.

The SFE Collaboration Working Group in FP7, chaired by the EU FP7 FAST project, together with the Networked European Software and Services Initiative (NESSI) User-Services Interaction (USI) Working Group, with the support of the Software and Service Architectures and Infrastructures Unit of the European Commission (SSA&I, D3, Mr. Jesus Villasante, Head of Unit), has recently been created to lead the innovation on service front-end technologies in the future internet of services.

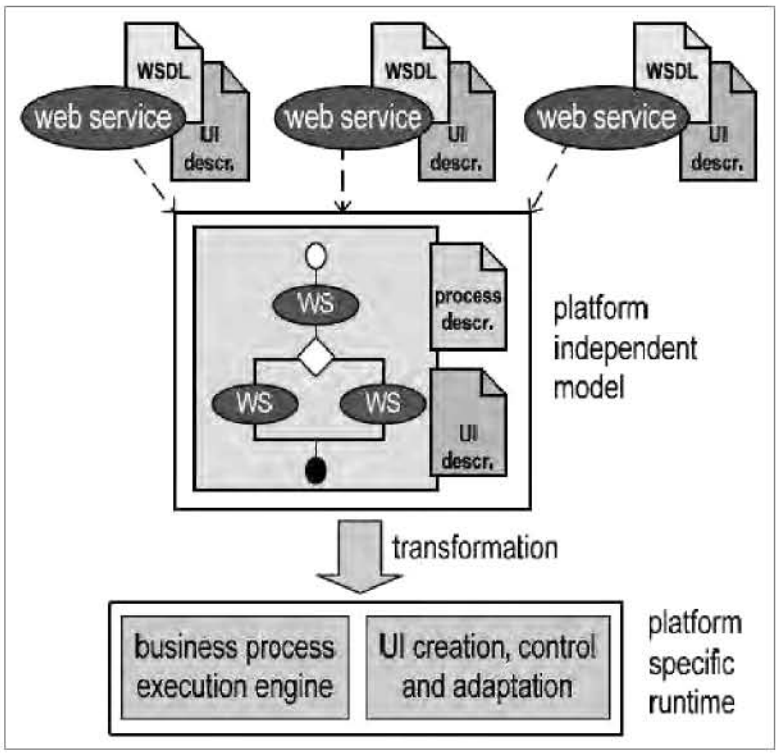
The objective of this group is to set up a collaboration schema among projects to effectively deliver a common vision of the service front-end technologies and architecture in the future internet of services.

The approach presented in this paper is part of the FAST initiative, a STREP project partially funded under the EU FP7 (INFSO-ICT-216048), as part of NESSI.

Other similar initiatives are beginning to proliferate in this research field. Of these, the NEXOF-RA project deserves special mention. We are active partners in this project. The project aims to build the Reference Architecture for the NESSI Open Service Framework by leveraging research in the area of service-based systems and aims to consolidate and trigger innovation in service-oriented economies. NESSI is the European technology platform dedicated to software and services.

Two projects (apart from FAST), called Servface and inContext, stand out in the SFE research working group.

Figure 20 MDA approach of ServFace (see online version for colours)



ServFace is a STREP project funded under EU FP7 that is partially related to the ideas presented in this paper. This project aims at creating a model-driven service engineering methodology for an integrated development process for service-based applications. ServFace will look at this process from two different perspectives:

1 the development of services with corresponding user interface descriptions

2 the development of user interfaces for a composition of services.

Comparing this project with the EzWeb/FAST approach, the key conclusion is that this initiative aims to add an integrated UI description and development approach to SOA concepts by introducing the notion of a corresponding user interface for services. This is a totally bottom-up approach: the idea is to enrich web services and resources with UI descriptions and build and integrate standard faces for this back-end into a global user interface. Therefore, it takes a completely opposite approach to this paper's top-down line of attack. ServFace aims to create composite applications from generic visual interfaces of the web services (rather than user-created UIs) that are then related to an existing enterprise back-end. For this reason, the resulting UIs would be more generic, less flexible and probably far removed from real user needs and requirements.

The inContext project is also closely related to the EzWeb and FAST projects. It is aimed at developing a novel scientific approach to the problem of enabling diverse individual knowledge workers in separate organisations to work in effective team collaboration with one another. The end result of the research will emphasise P2P collaboration software capabilities.

InContext will develop a novel scientific approach that is focused on a new blend of human collaboration and service-oriented systems. Therefore, inContext will explore novel techniques and algorithms for developing an understanding of the human activities involved. This whole approach is actually just a small part of the EzWeb/Fast approach for tackling web service composition and mashups, which ultimately also aims to foster and enable knowledge work and open innovation.

11 Conclusion and future trends

The appearance of user-centric approaches to next-generation service front-ends, such as the one proposed in this paper, will be a major step forward, providing solutions to currently hard-to-solve problems in the traditional WS-SOA paradigm. The emergence of such service architectures will solve key problems in three different scenarios. Large enterprises may capitalise on faster application de-velopment, a more agile system landscape and the empowerment of their employees to design their own applications that best satisfy their unique requirements, and to share this knowledge with other employees better than in traditional web service architectures.

On the other hand, the proposed architecture enables SMEs to find, customise, combine, catalogue, share and finally use applications that exactly meet their individual demands by leveraging the SaaS model, viewed as utopian from a traditional SOA perspective (Gartner Inc., 2006). Supported by the new internet of services approach, they can select and combine resources hosted by third parties rather than buying a predetermined, inflexible and potentially heavyweight solution (Winewright, 2005). Finally, individuals benefit from a strongly increased capability of personalisation and

participation. This approach will provide end users with intuitive, unsophisticated IT ways to discover, remix and use those web-based services that they consider interesting and useful. It will also allow them to participate, swap information with other users and service providers and to actively contribute in a way that encourages extensive use of the resources offered. This speeds up the service innovation pace. A user-centric SOA will involve the bulk of private users or small businesses and allow for 'customer self-service' (Hgg *et al.*, 2006). Future work will concentrate on evolving FAST and EzWeb, the open source composite application framework used as proof of concept in this paper. We expect them to become a major hub for the publishing, brokerage, customisation and, finally, consumption of web-based resources on a global, cross-organisational scale (Soriano *et al.*, 2007).

Acknowledgements

This work is supported in part by the European Commission under the first call of its Seventh Framework Program (FAST STREP Project, grant INFSO-ICT-216048) and by the European Social Fund and UPM under their Researcher Training programme.

References

Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004) *Web Services Concepts, Architectures and Applications*, Heidelberg, Germany: Springer Verlag, ISBN: 3-540-44008-9.

Anderson, C. (2006) 'The long tail, why the future of business is selling less of more', *Hyperion*, July.

Anderson, T. (2007) 'Is Yahoo Pipes all it's cracked up to be?', *IT Week*, 19 February, http://www.itweek.co.uk/itweek/comment/2185589/yahoo-pipes-cracked.

Bohl, O., Manouchehri, S. and Winand, U. (2007) 'Mobile information systems for the private everyday life', *Mobile Information Systems*, Vol. 3, Nos. 3–4, pp.135–152.

Coveyduc, J., Huang, C., Ostdiek, L. and Reif, J. (2006) 'IBM Innovation Factory. An integrated solution for accelerating innovation', *IBM*, October.

Davenport, T.H. (2005) 'Thinking for a living: how to get better performance and results from knowledge workers', *Harvard Business School Press*, Boston, Massachusetts, USA.

Dey, A.K. (2001) 'Understanding and using context', *Personal and Ubiquitous Computing Journal*, February, Vol. 5, No. 1, pp.4–7.

Driver, E., *et al.* (2004) 'Road map to an enterprise collaboration strategy', *Forrester Research*, 2 August.

Fielding, R.T. (2000) 'Architectural styles and the design of network-based software architectures', PhD thesis, University of California, Irvine.

Gartner Inc. (2006) 'Hype cycle for software as a service', *Gartner Research*, 10 August.

Hgg, R., Meckel, M., Stanoevska-Slabeva, K. and Martignoni, R. (2006) 'Overview of business models for web 2.0 communities', *Proceedings of GeNeMe 2006*, Dresden, pp.23–37.

Hierro, J.J., Janner, T., Lizcano, D., Reyes, M., Schroth, C. and Soriano, J. (2008) 'Enhancing user-service interaction through a global user-centric approach to SOA', *Proceedings of the Fourth International Conference on Networking and Services (ICNS 2008)*, IEEE Computer Society Press, March, pp.194–203.

IBM Developer Works (2006) 'Composite applications – business mash-ups', http://www.ibm.com/developerworks.

Lizcano, D., Soriano, J., Fernández, R., López, J.A. and Reyes, M. (2008) 'Tackling composite application developments from an enterprise 2.0 mash-up perspective', *Proceedings of the 14th International Conference on Concurrent Enterprising (ICE 2008)*, June.

MacKenzie, M. (2006) 'OASIS – reference model for service oriented architecture 1.0', http://www.oasis-open.org.

McAfee, A. (2005) 'Will web services really transform collaboration?', *MIT Sloan Management Review*, Vol. 46, No. 2.

McAfee, A. (2006) 'Enterprise 2.0: the dawn of emergent collaboration', *MIT Sloan Management Review*, Spring, Vol. 47, No. 3, pp.21–28.

NetVibes (2007) 'Universal Widget API specification', http://blog.netvibes.com/?2007/03/09/125-new-developer-website-preview-of-universal-widget-api-uwa.

North, D.C. (1990) *Institutions, Institutional Change and Economic Performance*, Cambridge: Cambridge University Press.

OASIS (2003) 'Web Services Composite Application Framework (WS-CAF) TC', http://www.oasis-open.org.

OASIS Open CSA (2007) 'Service Component Architecture (SCA)', http://www.oasis-opencsa.org/sca.

O'Reilly, T. (2005) 'What is web 2.0: design patterns and business models for the next generation of software', www.oreillynet.com/pub/what-is-web-20.html.

O'Reilly, T. and Musser, J. (2006) 'Web 2.0 principles and best practices', *O'Reilly Radar*, November.

Pilato, G., Augello, A., Vassallo, G. and Gaglio, S. (2008) 'EHeBby: an evocative humorist chat-bot', *Mobile Information Systems*, Vol. 4, No. 3, pp.165–181.

Pirrone, R., Russo, G., Cannella, V. and Peri, D. (2008) 'GAIML: a new language for verbal and graphical interaction in chatbots', *Mobile Information Systems*, Vol. 4, No. 3, pp.195–209.

Roman, D. (2005) *Web Service Modeling Ontology, Applied Ontology*, Vol. 1, No. 1, pp.77–106.

Schroth, C. and Christ, O. (2007) 'Brave new web: emerging design principles and technologies as enablers of a global SOA', *Proceedings of the 2007 IEEE International Conference on Services Computing (SCC 2007)*, p.8.

Schroth, C. and Janner, T. (2007) 'Web 2.0 and SOA: converging concepts enabling the internet of services', *IEEE IT Professional*, June, Vol. 9, No. 3, pp.36–41.

Smith, R. (2006) 'Enterprise mashups: an industry case study', Keynote at the *New York PHP Conference and Expo*, Manhattan, New York, USA, 14–16 June.

Soriano, J., Lizcano, D., Cañas, M., Reyes, M. and Hierro, J.J. (2007) 'Fostering innovation in a mashup-oriented enterprise 2.0 collaboration environment', *System and Information Science Notes, SIWN International Conference on Adaptive Business Systems (ICABS2007)*, Chengdu, China, July, Vol. 1, No. 1, pp.62–69.

Winewright, P. (2005) 'Why Microsoft can't best Google', *Software as a Service ZDNet Editorial*, August.